Yixue Zhang, Zheran Li

EC 504 Advanced Data Structure
Project Report

Boston University
Mechanical Engineering

Maze Solution with Multiple Path Finding Algorithms

December 12, 2018

1 Project Description

This project tested and compared performance and efficiency of BFS, DFS, Bellman-Ford, Wave-Front, Dijkstra and A* algorithms in an auto-generated maze. The generated maze has at least one way from starting node to goal node. With different maze sizes, we can see the performance is various based on work-space conditions.

2 Program Divisions

2.1 Maze Generation

The maze is an n by n square graph with a self-generate path from starting node to goal node. The rest part of this maze is building by randomly growth. The complexity and potential difficulty of the maze satisfies Gaussian distribution. We will run path finding algorithms in specific size mazes for multiple times so that we can reduce the testing error comes from the differentiated performance for a specific algorithm in a specific type of maze.

2.2 BFS, Wave-front and DFS Implement

The time complexity of BFS can be expressed as $O(|V| + |E|)$, since every vertex and every edge will be explored in the worst case. When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$, where $|V|$ is the cardinality of the set of vertices. The BFS algorithm in this project has a break condition detect if the goal node has been found. Once the path is

found, we don't have to iterate the rest nodes and arcs. DFS shares the same time complexity while this algorithm can't implement a break condition as BFS does. Wave-front algorithm is similar to BFS, the only difference is that wave-front search starting from the goal node and ending up at the starting node.

2.3 Bellman-Ford and Dijkstra Implement

BellmanFord proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. The approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path.

$$d(i, j) \leq d(i, k) + d(k, j)$$

The same as Bellman-Ford algorithm, Dijkstra also use this equation to update a closer vertex in the graph. However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman-Ford algorithm simply relaxes all the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman-Ford algorithm to be applied to a wider class of inputs than Dijkstra.

2.4 A* Implement

A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.
At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended.The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

3 Testing Result and Comparison

	Average Searching Steps	Average Running Time(ms)
BFS	1079	13.3728
DFS	1079	21.2363
Bellman-Ford	1079	1854.31
Wave-Front	1079	584.219
Dijkstra	1079	53.9173
A*	1079	42.4525

Table 1: Algorithms performance in 50 by 50 maze with average of 20 times run

	Average Searching Steps	Average Running Time(ms)
BFS	1605	14.3264
DFS	1605	149.141
Bellman-Ford	1605	29868.9
Wave-Front	1605	6030.82
Dijkstra	1605	143.041
A*	1605	116.849

Table 2: Algorithms performance in 100 by 100 maze with average of 20 times run

From the result tables above, we can see that in simple work-space, the five algorithms except Bellman-Ford have similar performance and efficiency. In larger and more complicate maze, the performance of BFS has a significant advantage compared with the rest algorithm. From here, we

	Average Searching Steps	Average Running Time(ms)
BFS	9	0.260926
DFS	9	0.246555
Bellman-Ford	9	0.843464
Wave-Front	9	0.297089
Dijkstra	9	0.190777
A*	9	0.357041

Table 3: Algorithms performance in 5 by 5 maze with average of 20 times run

can say that a complicate large scale work-space will more likely works as a path finding algorithm's worst case, and BFS has the best working efficiency among the others in such cases. While in average or best cases, Dijkstra has a better efficiency due to its more effective path update strategy.